ENDOR LABS

# State of Dependency Management

STATION 9

BY ENDOR LABS

# Open Source - Here, There and Everywhere!

The software industry is in turmoil. Vulnerabilities in widely used open source components continue making headlines, and so do an increasing number of software supply chain attacks, where adversaries try to sneak malicious code both onto developer and end-user systems. Remarkable incidents include the now infamous Log4Shell vulnerability disclosed in December 2021, and protestware published in March 2022, where open source maintainers weaponize their own project to express a political opinion.

One key reason for this excitement is non-technical, and not driven by specific attacks on or vulnerabilities in a given piece of software. It is the increased attention of government and regulatory bodies on the subject of software supply chain security. Starting in 2020, we see a number of regulatory efforts aiming to guide how software development organizations consume open source and 3rd party software.

The **basic** assurance level of the candidate EU Cybersecurity Certification Scheme for Cloud Services, for instance, requires that cloud service providers "**shall maintain a list of dependencies to hardware and software products used in the development of its cloud service.**" [1] The next assurance level expects service providers to develop and implement policies around, for instance, component age or updates. And on this side of the pond, following the White House Executive Orders from February and May 2021, we're seeing a wealth of guidance coming from the NIST, DoC and NTIA around topics such as software bills of material (SBOM) and supply chain security in general.

This attention is overdue and will hopefully lead to a situation comparable to the manufacturing industry, such that we as a society have visibility into the quality, security, safety and provenance of software components that are of critical importance in our daily lives, spanning everything from critical infrastructures to autonomous vehicles.

Of course, the industry cannot simply offload security requirements on the backs of spare-time open source maintainers. Thankfully, a number of non-profit organizations like the OpenSSF or OWASP set out to support critical open source ecosystems, supported by major players in the software industry.

So what does this mean for institutional development organizations from the private or public sector? First, that they need to prepare for emerging regulations, which requires digging deep into the dependency relationships of open source packages, also referred to as 'dependency hell' [2]. Non-compliance can result in direct negative impacts to businesses, even without a breach ever taking place; for example, when companies fail to produce software bills of material of sufficient quality for the public sector.

And in addition to security concerns, they'll also need to tackle operational risks resulting from the consumption of 3rd party and open source components. This requires monitoring non-functional properties, e.g., the liveliness of open source projects, during the entire dependency lifecycle, starting from the initial selection and inclusion of a dependency and throughout its entire lifetime.

With this report, the first but certainly not the last of its kind from Endor Labs, we'd like to share insights on the intricacies of modern, open source-based software development, and provide guidance on what matters most for software developers.

# Selected supply chain security events

(June 2020 - now)

community event | regulatory event | Incident

**Jun 2020**

OWASP Software Component Verification Standard

**Dec 2020**

Candidate EU Cybersecurity Certification Scheme for Cloud Services

**May 2021**

President Bidens 2nd Executive order 14028

**Feb 2021**

President Bidens Executive order 14028

**Jun 2021**

OpenSSF SLSA

**Dec 2020**

Disclosure of SolarWinds

**Jul 2020**

ENISA Threat Landscape for Supply Chain Attacks

**Jan 2021**

Faker and colors

**Dec 2021**

Log4Shell

**Jul 2020**

NTIA Releases Minimum Elements for a Software Bill of Materials

**Mar 2021**

Protestware Node-ipc, styled-components, es5-ext

**Feb 2022**

OpenSSF Alpha & Omega

**Oct 2022**

Vulnerability in Apache Commons Text

**Sep 2022**

Securing Open Source Software Act of 2022

# Methodology

Addressing above-described supply chain security risks requires an understanding of the state of dependency management in modern application development. To this end, we chose the Census II report[3] as a starting point, a data set meant to contain “the most widely used FOSS deployed within applications by private and public organizations”. Published in March 2022 by the Linux Foundation and Laboratory for Innovation Science at Harvard, it’s been created on the basis of scan data provided by several commercial Software Composition Analysis (SCA) vendors.

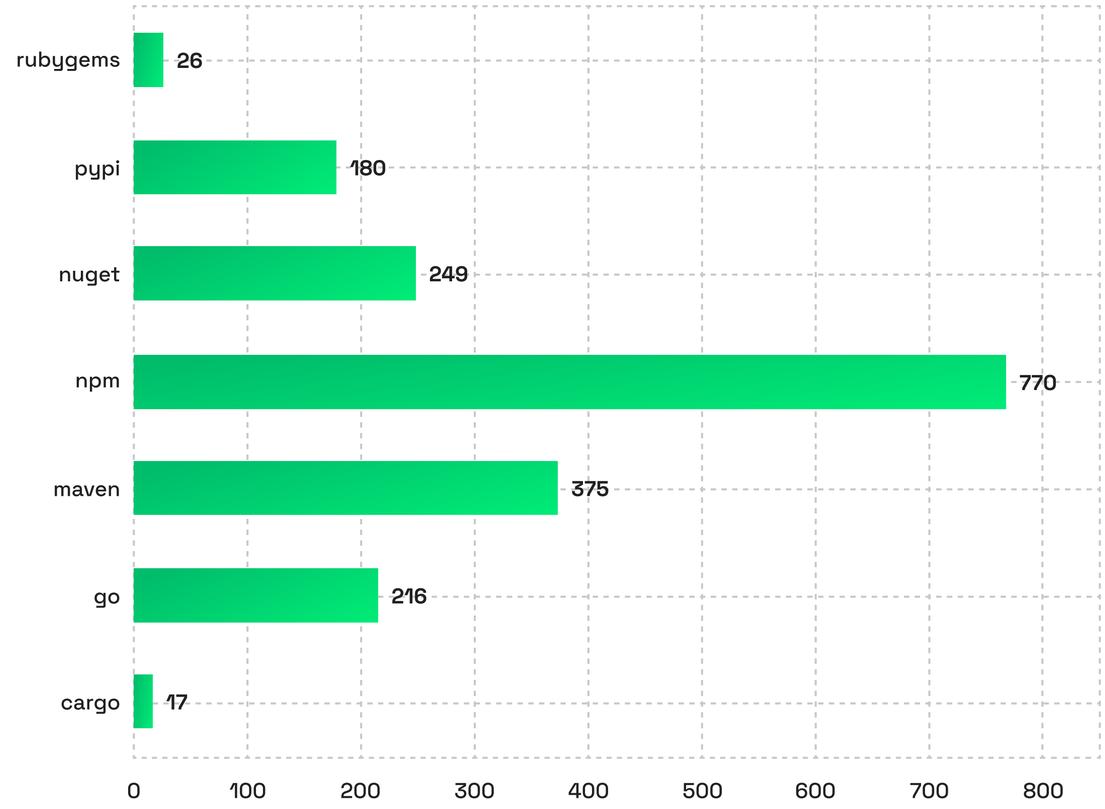
The initial data set – a total of 1833 distinct packages mentioned in Census II appendices A-H (excluding ecosystems represented with less than 10 packages, cf. chart 1) – is further enriched by several other data sources and tools. Libraries.io provided the location of the source code repository for a subset of 1413 packages, 1377 of which could be called successfully via HTTP , which is required to query Coreinfrastructure.org (for OpenSSF Best-Practices Badges), to run the tools developed by the OpenSSF projects Security Scorecards and Criticality (current scores and ratings), and to query Google BigQuery (historical scores and ratings).

To dig deeper into the dependency graphs of 375 distinct Java packages, we used Maven and Maven Central (to determine dependencies, recent releases and release dates), as well as OSV and NVD (to find vulnerabilities, CVSS ratings and vulnerability disclosure dates). In the end, we obtained release information for 356 Maven packages, and dependency graphs for 351 of them.

The enriched Census II data set as well as the dependency and vulnerability information for Maven packages is available for download on our GitHub [4] so that others can replicate, update and extend our analyses.

Chart 1

Distribution of the 1883 Census II packages used for this report (excluding ecosystemswith <10 packages)




Box plots will be used at several places in the document to show statistical properties of numerical data. They visualize data through a box and so-called whiskers. The box is defined through the quartiles Q1, Q2 (also called median) and Q3, which split the data values into four equally-sized groups such that 25% of the values are below quartile Q1, another 25% between Q1 and Q2, etc. The whiskers show the smallest and the greatest value respectively. Finally, the average or mean of the values is illustrated using a green triangle.

# The community starts securing critical projects… Do they cover the ones important for you?

As consumers and maintainers of open source software realized the risks associated with its use, several initiatives started identifying critical projects so they could receive adequate support – support that reflects their importance to the software industry.

The Census II report is one such example, based on actual scan data of production applications. Another is the OpenSSF Criticality Score [5], which computes a score for any Git repository hosted on GitLab or GitHub, solely on the basis of publicly available information such as age or the number of contributors.

Comparing those initiatives, however, reveals that **it is anything but straightforward to develop an algorithm that determines general project criticality:** Many projects highlighted by Census II receive relatively low criticality scores (cf. chart 2), suggesting that the parameters and weights require further fine-tuning to better capture projects used by applications developed and operated by private and public organizations.

This observation is confirmed by the little overlap between the Census II projects and the Top-200 GitHub projects of the respective programming languages [6] (cf. charts 3 and 4 for Java and JavaScript).

The Alpha-Omega project [7] considers both Census II and OpenSSF Criticality Score - which will help to provide coverage to many projects. Initiatives such as those from the OpenSSF go a long way to determine which OSS projects are “critical”, but **OSS consumers keep overall responsibility and need to address security risks according to their specific circumstances, e.g., deployment model, legal and contractual obligations or risk appetite.**

Chart 2

Is it critical? 75% of Census II packages have a relatively low criticality score - under 0.64

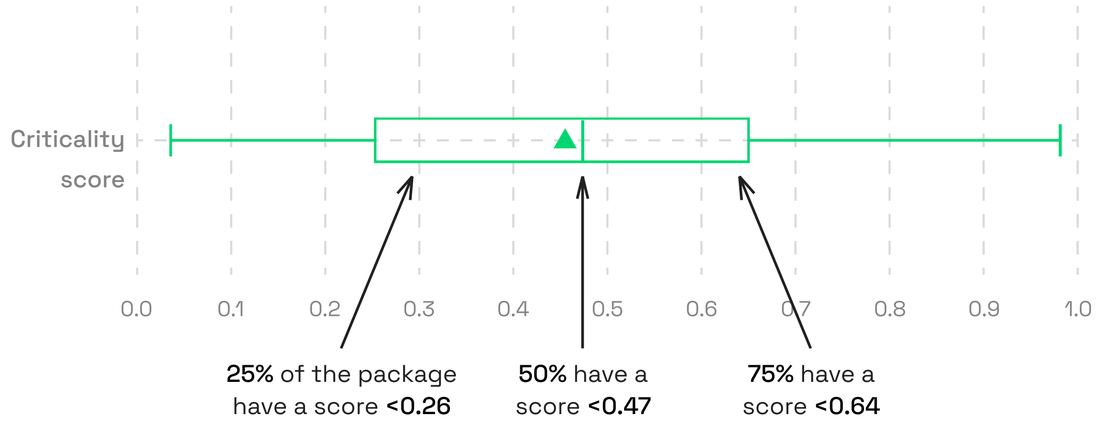
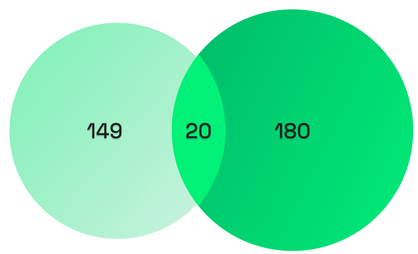



Chart 3

Small overlap between Census II and OpenSSF criticality Top 200 - Maven

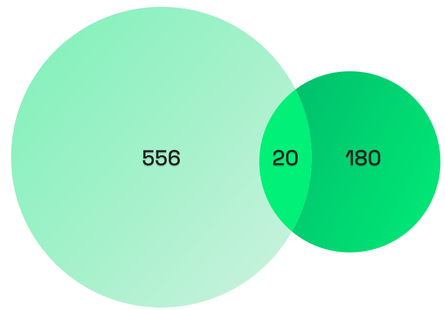



Distinct GitHub repos of 264 Maven packages in Census II

Top-200 Java repos from OpenSSF Criticality

Chart 4

Small overlap between Census II and OpenSSF criticality Top 200 - JS




Distinct GitHub repos of 695 npm packages in Census II

Top-200 JavaScript repos from OpenSSF Criticality

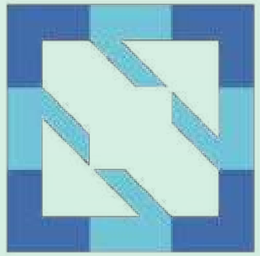
**The OpenSSF provides tools and guidance to help strengthen the security of major OSS projects. However, organizations consuming OSS are responsible for addressing security risks according to their specific circumstances, and must evaluate for themselves - what is critical.**

# Open source security - A Partnership

In the early days of cloud adoption, the concept of the “shared responsibility model” for cloud security emerged. According to this model, the Cloud Service Provider (CSP) owns the security of the underlying infrastructure on which customer workloads run, and customers own the security of the workloads themselves. So while the CSP owns security for the physical servers, networks, hypervisors etc., the customer owns the security of their code, data, images, and anything they run on the services provided by the CSP.

In the world of open source software, the model would be a different. When an organization chooses to rely on an open source project, they take responsibility for how that code might affect their security posture. While many maintainers hold their own projects to high security standards, they have no obligation to do so as per the license terms. Many of them freely contribute their time and skills to build projects that help thousands of companies across the world.  Fortunately, open source foundations like the OpenSSF, CNCF or OWASP, just to name a few major players, step up to secure the way today’s software is built, e.g., by advocating security best-practices or developing security tooling.

So now, effectively, the version of the “shared responsibility model” in open source security is a trust-based partnership between the organizations that rely on OSS, and initiatives such as the OpenSSF, CNCF and OWASP - who strive to improve the overall security posture of major open source projects. And as always, organizations need to be respectful and remember we are walking on the shoulders of giants, many of the most used open source projects are thanklessly maintained by individuals. Tools like Scorecard, and the upcoming Alpha & Omega go a long way in identifying critical open source projects and improving their security posture. It is now up to the organizations consuming open source to participate in these projects, and understand how the projects they use impact their overall security posture.


# Risk Indicators - The road so far

Anybody who has ever selected a commercial or open source component to be included in a production application most likely had to check certain properties beforehand. Functional properties and license compliance aside, **open source components should meet several quality criteria in order to reduce the operational, quality, and security risks resulting from their inclusion.** This need and common selection criteria have been described by academia and industry [8] [9] [10], and such criteria, of course, need to be monitored throughout the entire dependency lifecycle.

Shortly after the Heartbleed vulnerability in 2014, the Core Infrastructure Initiative (CII) acknowledged these needs and initiated the Best Practices Badge program [11]. Open source maintainers can self-assess projects in terms of quality, security and other factors, and receive passing, silver or gold badges. A more recent effort dedicated to security, the OpenSSF Security Scorecards, does not rely on self-assessments but rather on the automated evaluation of Git repositories to assess their security posture.

Because of the prevalent need to evaluate the security of upstream open source, ideally in a quantifiable way, we found it worthwhile to study the presence and evolution of badges and security ratings of packages that are part of Census II, both of which are very visible in and sponsored by the open source security community.

**Unfortunately, the adoption of best practice badges did not improve**. From the perspective of developers with hundreds of dependencies, only a small fraction are covered by the badge program (47 participate, 12 with passing badge). A probable reason for the low adoption is that maintainers need to proactively provide a self-assessment – ideally on a regular basis, which is a tough requirement to place on voluntary 'spare-time' contributors. Despite their best intentions, the workload associated with project maintenance and development often surpasses the time they can spend, all of which **speaks in favor of automated ratings à la scorecards**.

**The good news is that the scorecard ratings improved** between Sep 2021 (up until which historical data is available), and today (cf. charts). The average scorecard rating went up from 4.3 to 5.3, the minimum from 1.6 to 3.3, and the maximum from 8.1 (go-genproto) to 9.2 (urllib3).

Chart 5 Historical OpenSSF Scorecard ratings of Census II packages

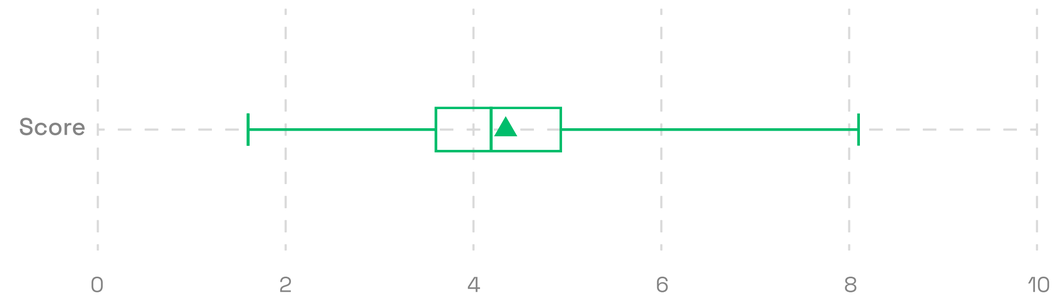
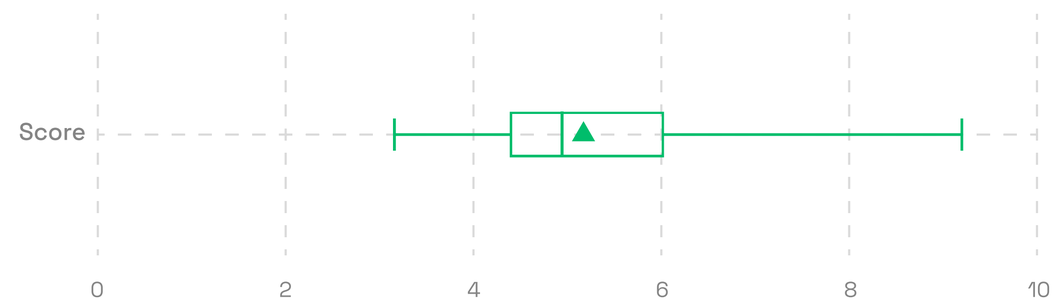



Chart 6 Current OpenSSF Scorecard ratings of Census II packages - improvement over time!




**The automated method of evaluating the security of open source packages has won, and scores have improved over time!**

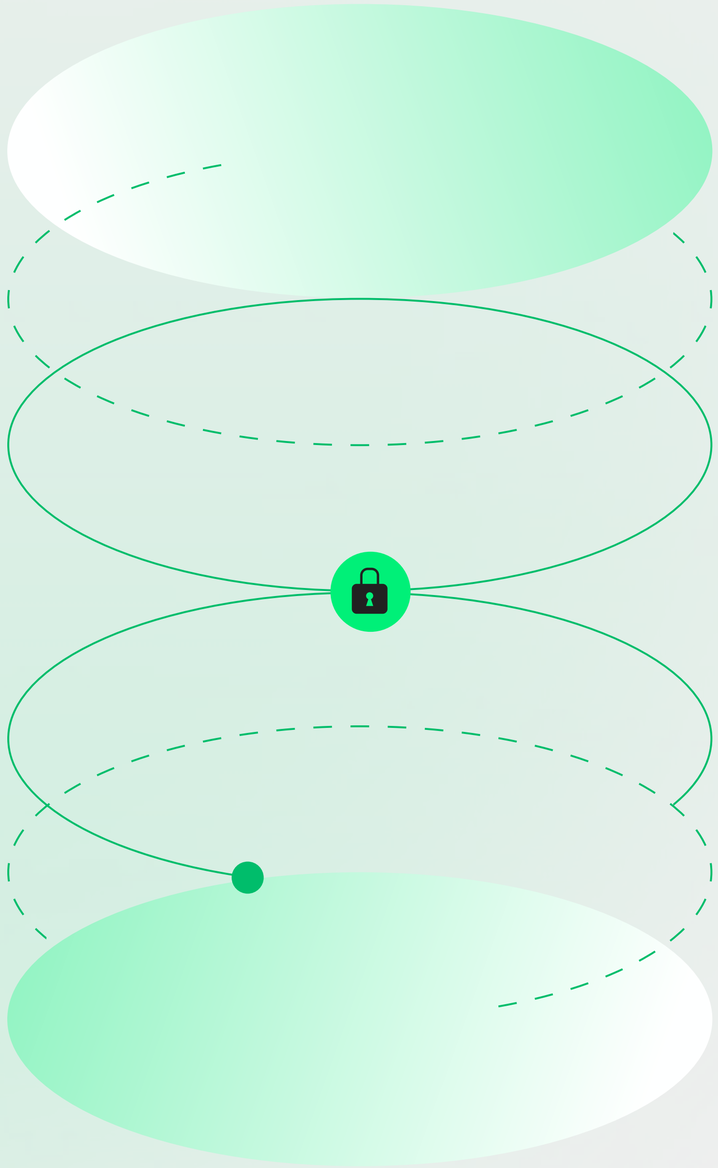
# Risk Indicators - The road so far

Monitoring such metrics is essential to implement current and upcoming guidance, e.g., the recommendations to check for outdated or end-of-life components and projects[12] [13], but can hardly be done manually, without tool support.

The broader the scope of such metrics, the better consumers can take risk-based decisions whether or not to consume a given open source project. For example, now that some ecosystems have started to enforce 2FA for maintainers of critical components, it seems natural to extend the scores accordingly, and thereby cover one of the attack vectors[14] that is prominently[15] used for supply chain attacks on legitimate projects.

Metrics can also support the detection and mitigation of name confusion attacks[16] - the infamous typo-squatting, brand-jacking and a number of comparable techniques, used by attackers to confuse or trick developers into installing a malicious package rather than the intended one. To this end, attackers commonly create new projects and packages with similar names than legitimate ones (e.g., mumpy instead of numpy). But apart from the name, they differ significantly in other regards, e.g., age, number of contributors, commits or releases, which can be discovered in an automated fashion using metrics that look at project activity or popularity.

However, as important as they are, such risk indicators **will not be able to protect against all of the recent supply chain attacks.** The dependency confusion attack* [17] [18] , for instance, exploits weaknesses in the dependency resolution process of end-users' package managers or internal repositories, which is out of scope of quality indicators for open source projects. Unfortunately, dependency confusion attacks were among the most widely used attack vectors, including massive campaigns with hundreds of malicious packages being deployed in some automated fashion[19].



* Where consumers believe consuming trustworthy internal projects, while a vulnerable resolution process picks malicious ones from public repos.

# Dependencies are more complex than you think

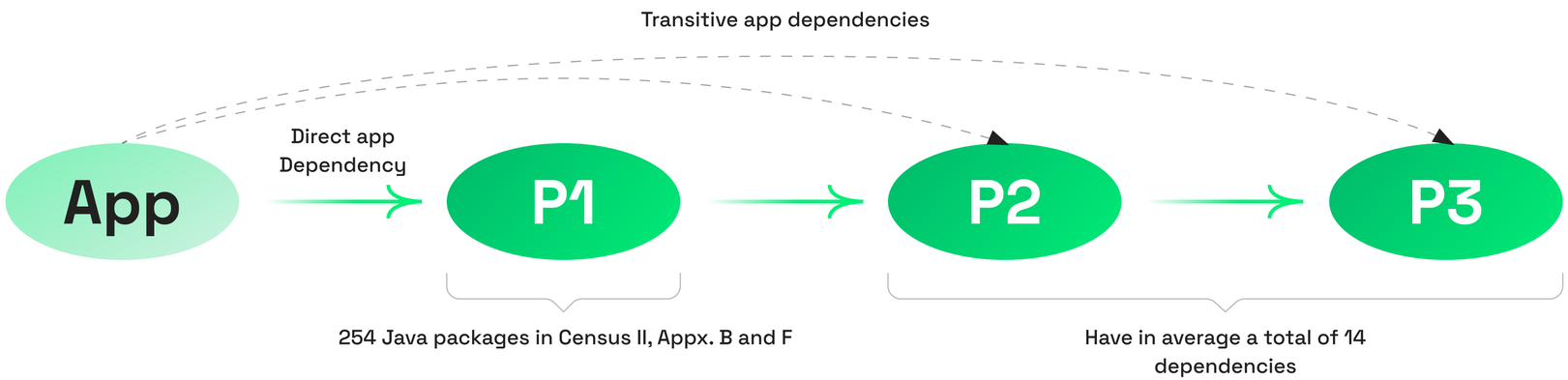
Dependency relationships between open source packages are intricate – this has been shown in numerous academic studies[20][21] , and is no different for the Java packages mentioned in the Census II report. A component depends directly on another component if there's an edge between their corresponding nodes in the dependency graph (solid edges in chart 8). In the case of transitive or indirect dependencies, two components are only connected through other ones, e.g., application App transitively depends on P2, thanks to the direct dependencies of App on P1 and P1 on P2 (dotted edges).

The 254 distinct Maven packages mentioned in Census II Appendices B and F are all direct dependencies of applications developed by public or private organizations, comparable to P1. All of their dependencies are transitive dependencies from the perspective of the application developer - considered "internal affairs" of P1, many of them pulled automatically into the application development project by Maven or other package managers.

The latest versions of the 254 packages analyzed have an **average number of 14 dependencies (direct and transitive)**, which is less than what has been reported for other ecosystems[22], esp. npm[23] [24] , whose inflated dependency trees are due to the phenomenon of micro-packages[25]. However, considering that a typical application declares several direct dependencies, they easily end up with a total of dozens if not hundreds of dependencies[26]. The average depth of those dependency trees is 2, just like the distance of P1 and P3 in the example, and the maximum depth is 7.

Six outliers stand out with more than 100 dependencies each. Among those are **aws-java-sdk v1.12.327** from Oct 24, 2022 with a total of **331 dependencies** and **log4j-core v2.19.0** from Sep 13, 2022 with **141 dependencies.** Even though the majority of those is used for testing - **who would have thought before Log4Shell that logging comes with such complexity?**

Chart 8 | Explaining direct and transitive dependencies




The **lastest** versions of 254 Maven packages had **14** dependencies on average, with outliers like:

aws-java-sdk v1.12.327 **with 331 dependencies**

log4j-core v2.19.0 **with 141 dependencies**

# Dependencies are more complex than you think

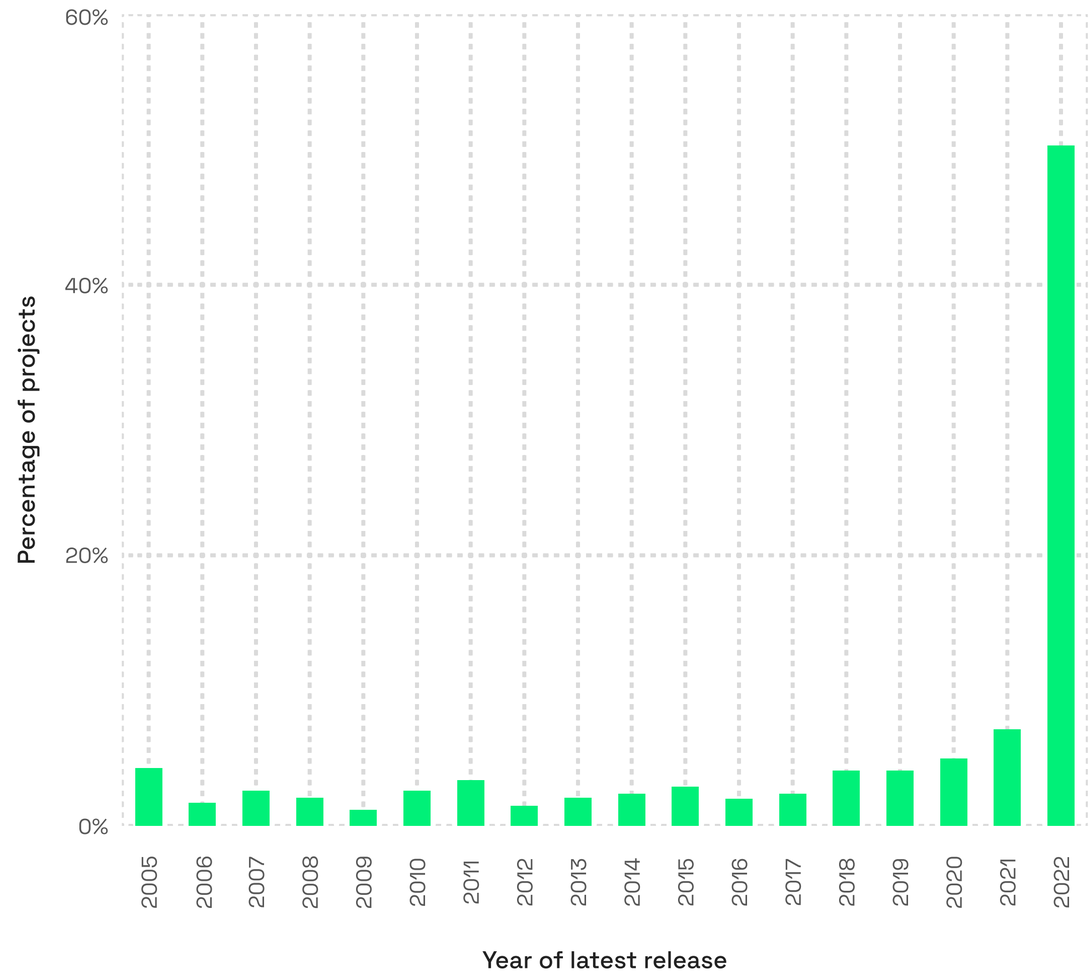
When it comes to assessing vulnerabilities in any of those packages, it is important to know which ones will be deployed together with the application and which not. Dependencies only used for testing, for instance, will not end-up in production, hence, any vulnerabilities affecting those can be deprioritized. **SCA tools that miss this context cause hundreds of wasted hours patching non-critical vulnerabilities.**

13 out of 356 most-used Census II Maven packages have a latest release identifier starting with 0, e.g., perfmark-api v0.25.0 from Feb 25, 2022. According to semantic versioning[27] (semver), a widely-used versioning scheme, such pre-releases are to be considered in "initial development, where anything may change at any time". This observation is inline with others suggesting that developers do not strictly follow semver, both in regards to the (unexpected) stability of pre-releases[28] and the (expected) backward compatibility of minor releases[29]. **Using program analysis techniques more thoroughly, e.g., for update recommendations, will reduce update regressions**, no matter the release identifier or versioning scheme used.

It is also interesting to know whether those 241 packages are still actively maintained. **Software ages like milk, not like wine, as packages without recent releases risk containing many vulnerable dependencies**, potentially exploitable in the context of downstream applications. Chart 9 shows that more than 50% of the Census II packages have a latest release in 2022. Whether those are used is another question though... and even if they are relatively fresh, it still does not mean they are free of vulnerabilities.

**50% of the most used Census II packages didn't have a release in 2022, and 30% had their latest release before 2018 - these can cause serious security and operational issues in the future.**

Chart 9 — Census II packages and year of their latest release



# Software is like milk - It gets sour quick

One result of our analysis is that even though many Census II Maven packages have a latest release from 2022, many of those have one or more vulnerable dependencies (cf. pie chart), which is inline with other studies[30] [31] . If you pick the latest release of any of the Maven packages in Census II, **there's a 32% chance it will have one or more known vulnerabilities hidden in its dependency tree**. Among the outliers is archaius-core v0.7.7, released on Sep 5, 2019, with 49 vulnerabilities in its dependencies. Archaius is still maintained, however, the current version 2.3.17 is deployed using a different name (archaius2-core), and the migration of users to this 2.x release will be hindered by its backward incompatibility[32]. **Again, this is an opportunity for program analysis techniques to detect and remediate code-level incompatibilities.**

In the Java world, this phenomenon is likely due to the fact that **all components on the path from a user to the vulnerable component need to update the version identifier in order to have the user consume a fixed, non-vulnerable version**. Unless all components on the path use version ranges (which is uncommon in Java) or consumers override a version by adding a direct dependency (which jeopardizes the original idea of automated dependency management). Those work-arounds aside, if P3 in Chart 8 had a vulnerability, P2 and P1 had to produce new versions as well so the application could benefit from a fix. To support consumers in fixing vulnerable transitive dependencies SCA tools should not content themselves to recommend "update x to y", but **find the closest possible fix in the application's specific dependency path**.

In total, roughly **95% of the vulnerable dependencies are transitive ones from the perspective of the application** (level 1 or greater in the bar chart), which makes it very difficult for application developers to assess whether a given vulnerability in such transitive dependency is indeed reachable and exploitable in their application context. Such assessments are required for the simple reason that not all of the code contained in the many components pulled into a development project is actually executed and needed in a given application. The recent vulnerability discovered in Apache Commons Text[33], for instance, only matters for an application if the vulnerable class StringSubstitutor is used (by the application itself or in any of its dependencies). **But how could a developer ever know whether that is the case for hundreds of dependencies? Again, program analysis can come to the rescue**, and help implementing recommendations to remove unused dependencies[34].

**Chart 10** 25% of 179 packages that have a release in 2022, still have between 1-18 vulnerable dependencies!

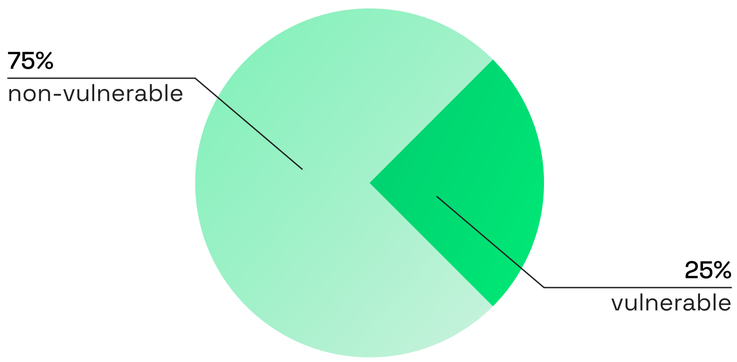
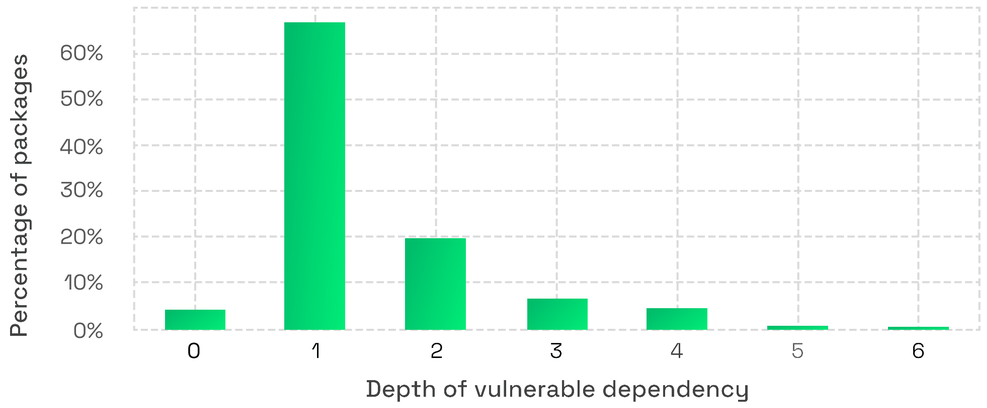



**Chart 11** Where in the dependency tree are vulnerable dependencies




**95% of vulnerable dependencies are transitive, and when you update, there's a 32% chance you will still have vulnerabilities.**

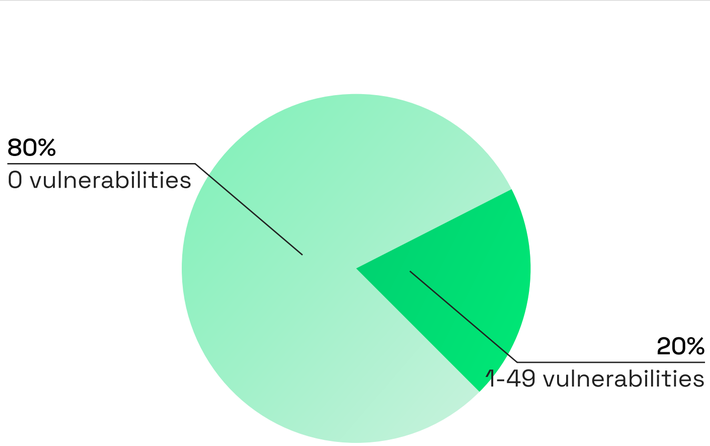
# How to prioritize vulnerabilities?

New vulnerabilities are disclosed almost on a daily basis - This makes prioritization crucial. One criterion is the type of the dependency. Test dependencies, for example, are only employed during unit or integration tests. Being excluded from the productive application, vulnerabilities therein can largely be ignored. According to one study, “**about 20% of the dependencies affected by a known vulnerability are not deployed**”[35] . However, **even when ignoring test dependencies altogether, the share of packages whose latest release contains one or more vulnerable dependencies only drops from 32% to 20%.** In other words, the latest release of every fifth package in the Census II report still brings in vulnerable dependencies (see pie chart 12).

Another criterion used for prioritization are CVSS scores, which consider different metrics such as the attack complexity or the potential impact in order to estimate the severity of a given vulnerability. Additional metrics can be used to reflect the availability of exploits, the presence and maturity of fixes, or environment-specific properties to adjust the overall severity. But the crux with CVSS scores is two-fold: Vulnerabilities with CVSS scores greater than 7 (high and critical in bar chart 13), a threshold often triggering urgent mitigation, represent the majority of findings. But more importantly, **vulnerabilities with high CVSS scores are not necessarily the ones being actively exploited**[36] [37].

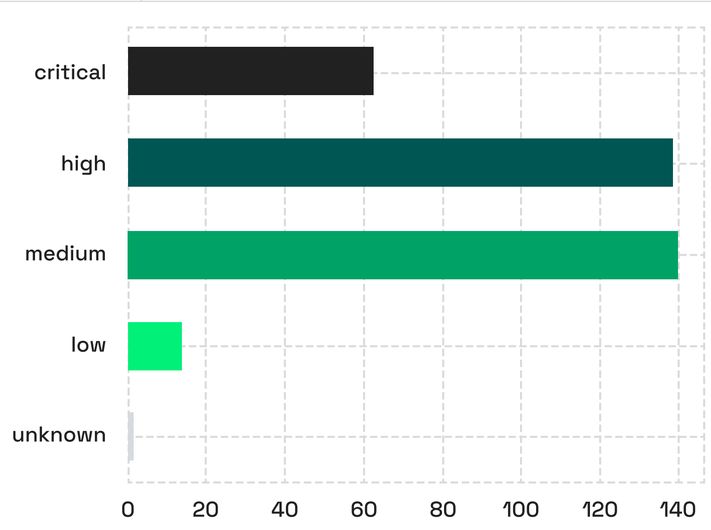
One criterion we recommend for prioritization is the reachability of the vulnerable code. As discussed in the context of CVE-2022-42889[38], the vulnerability affecting Apache Commons Text, what matters is whether the vulnerable code can be reached in the context of a given application. **Reaching vulnerable code is a prerequisite for exploitability**[39], and early studies of the phenomenon of software bloat in Java applications indicate that **a significant share of code pulled into a project is not used at all in its context, sometimes entire packages**[53] [54]. Continuing the example introduced before, vulnerable methods in P2 and P3, highlighted in red, can be fixed with lower priority if they cannot be reached from the application code.

Chart 12 — Do the latest releases of 351 packages have vulnerable dependencies (exluding test scope)?



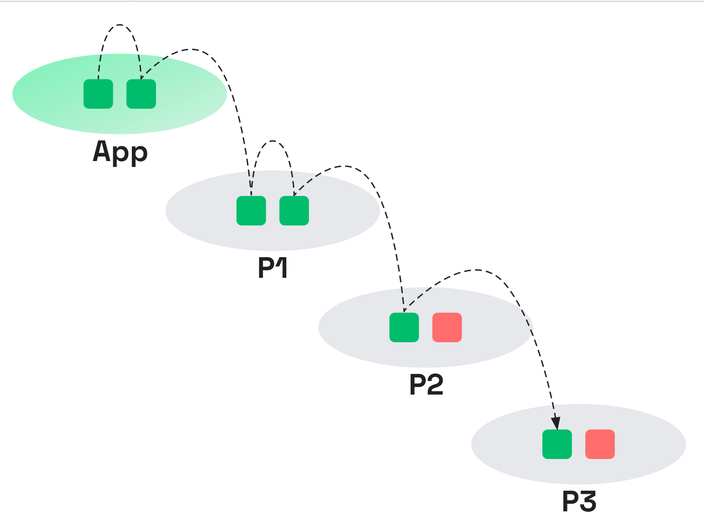

The latest release of 32% of packages (across all years) have vulnerable dependencies, ignoring ”test” dependencies only takes it down to 20%.

Chart 13 — Criticality of vulnerabilities in the dependencies of latest versions of 351 Census II Maven packages




Using CVSS scores, most of the vulnerabilities are ranked high or above - so not much help prioritizing.

Chart 14 — Understanding reachability




Understanding reachability: The red boxes are vulnerable methods within dependencies, if they are not invoked - they should be deprioritized.

# So, do we just update?

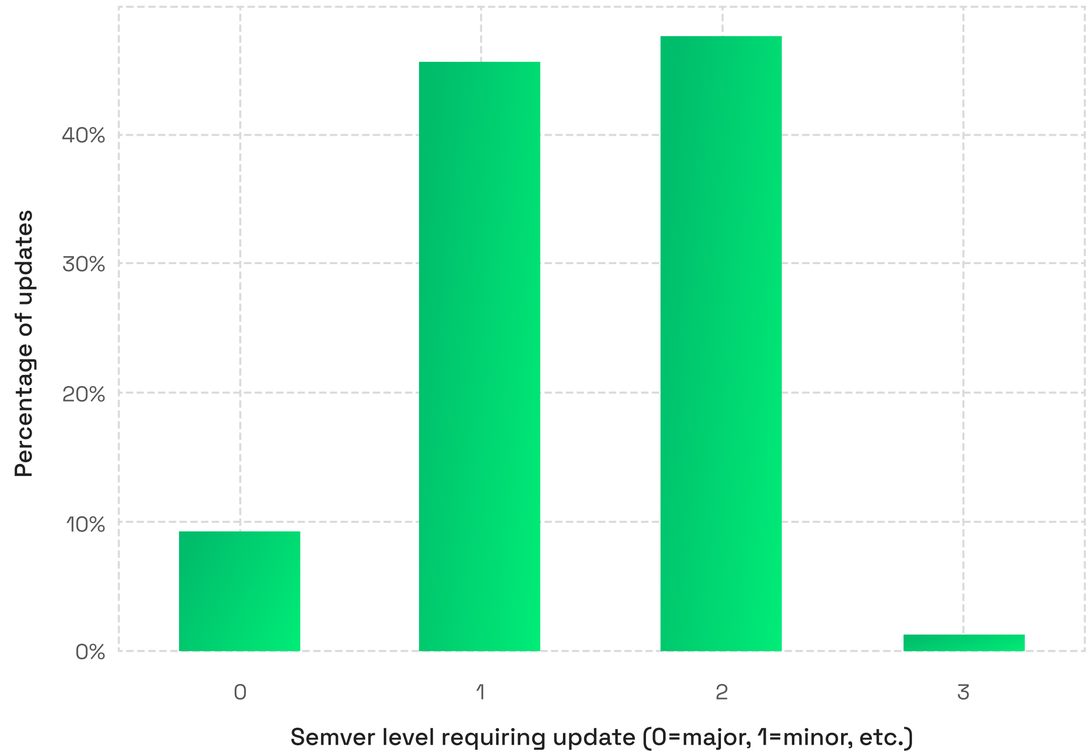
It is well-known that **timely patches are key** to prevent systems from being exploited. One study showed that 37% of 11,079 public exploits in Exploit Database were available before (0-days) or within one week after a patch was released, and 80% of those were available before the corresponding CVEs were published[38]. Another observed that **"75% of exploit code is observed within 28 days"**[39]. CVE-2017-5638, a severe vulnerability in Apache Struts that led to the Equifax data breach, is another example demonstrating that public vulnerability disclosure and large-scale, automated exploit attempts happen within a range of a few hours only[40]. One possible explanation for fast exploit availability may be that attackers monitor open source code repositories to discover security fixes (and with that, also the vulnerabilities), which happen often before the actual patch is available for download or the vulnerability is publicly disclosed[41].

Let's start by assuming that a patch, thus, a non-vulnerable version of the affected package is available, thus, a developer could update from version X to Y. But **is that new version compatible with the old version**, does it have the very same API so that the application code can remain as-is? And if it has, does it also behave in the same way? According to the semantic versioning scheme, only major version updates are backward-incompatible, whereas minor and patch updates remain compatible.

Chart 15 shows the version updates necessary to fix vulnerabilities in the latest versions of the Census II Java packages. About 9% of the updates require a change of the major version, which are per semver definition backward-incompatible, thus, require changes of the application code. This typically happens if projects remain on old and unmaintained releases, which is why developers should try staying on supported releases, to avoid risky updates during urgent security response processes.

Many updates, roughly 44%, require a change of the minor version, which indicates semver compatibility. However, a study of the Maven ecosystem found that there's no difference in the number of breaking changes introduced by major and minor releases[42]. A more recent study finds that 20% of non-major releases are breaking[43]. **Static software analysis can support developers in such cases through the automated analysis of application programming interfaces, and suggesting those versions that come with the smallest likelihood of regressions.**

Chart 15 — How many dependency updates require changes of the major, minor, etc.? (computed over 2912 vulnerability updates)




Be careful with updates: 9% of vulnerability fixes will require a major update, which by definition will add a breaking change. 44% will require a minor change, but studies show that since not everyone follows semver very closely, these updates are **just as likely to cause issues.**

# So, do we just update?

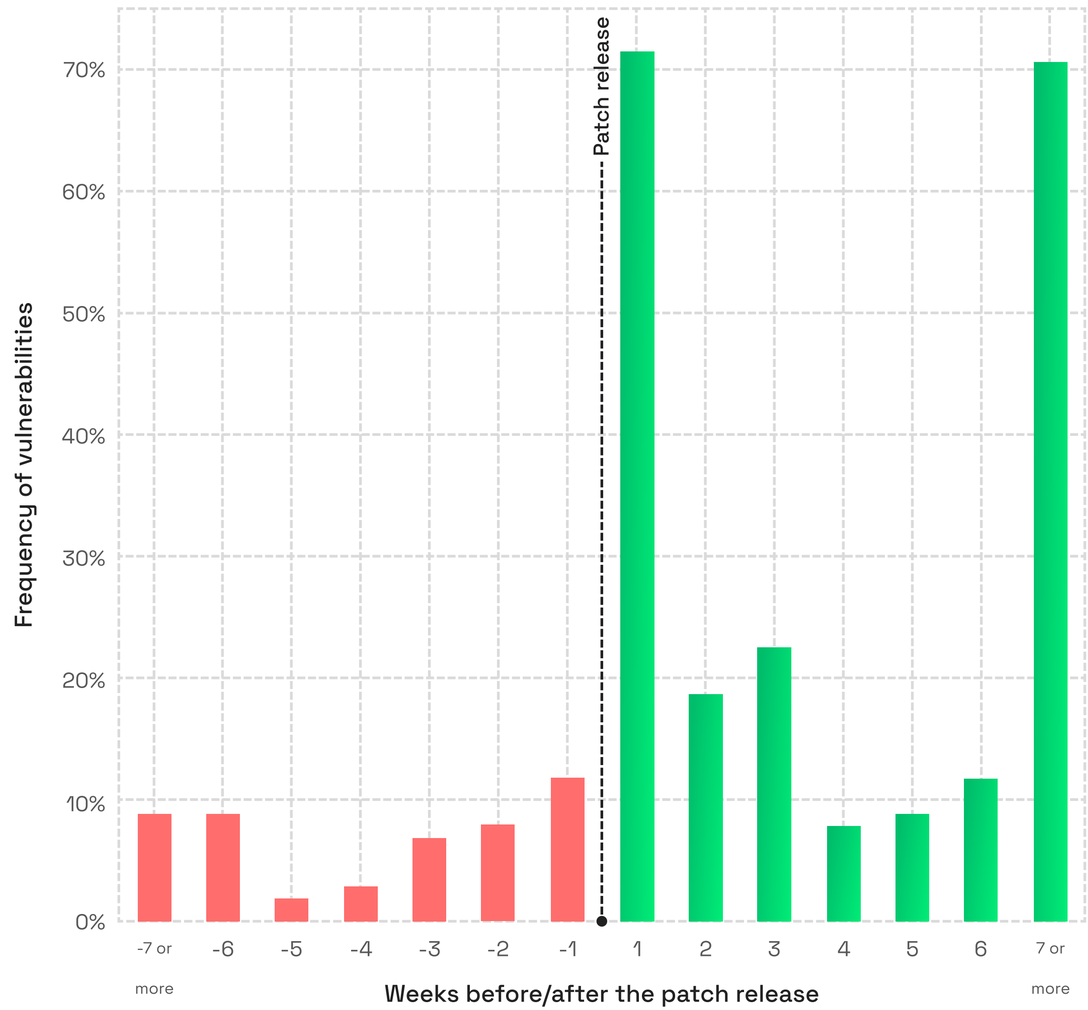
A particular case of updates are the so-called semantic updates: those are not API incompatible but modify the updated code's contract, for example, by fixing bugs that change the results returned for the same input. While, in theory, tests should be able to catch those changes, a recent study of 500 OSS Java projects found that tests only cover around 50% percent of calls in direct dependencies and only 20% of the calls in transitive ones. The same study advocates **static analysis as a way to mitigate the issue of semantic updates**, raising the number of detected semantic updates to 70% on average.

But what if no patch is available yet? Chart 16 shows how many weeks lie between CVE vulnerability publication and the release of a corresponding patch (knowing that "NVD, as an aggregator, typically publishes CVEs after advisories have been announced at other platforms"[44]). Numbers highlighted in red indicate vulnerabilities that were published in the different weeks before the patch was available, those in green were published in the weeks after patch release. The observation that a non-negligible number of patches are released after vulnerability publication is inline with a recent study of the npm, Maven and Nuget ecosystems[45].

Spot checks show that some of the delayed patches are due to maintainers not agreeing on the reported finding being a real vulnerability, or considering the severity exaggerated (e.g., CVE-2016-1000031[46] or CVE-2022-38752[47]). Ideally, such conflicts are resolved prior to findings being disclosed, which either exposes users (in case of real vulnerabilities without patch) or wastes developer time (to chase after non-vulnerable findings reported by SCA tools).

**Just updating to the latest version may introduce unexpected comparability problems, and isn't always an option. Static analysis can help determine the best course of action.**

Chart 16 — Were vulnerabilites disclosed before or after a patch was available?



# Supply chain attacks - As if vulnerabilities were not enough

Known vulnerabilities in upstream open source dependencies is not the only worry of application developers. After early research on typosquatting in 2016[48] and occasional typosquatting attacks in the years to follow, the frequency of supply chain attacks keeps on increasing.

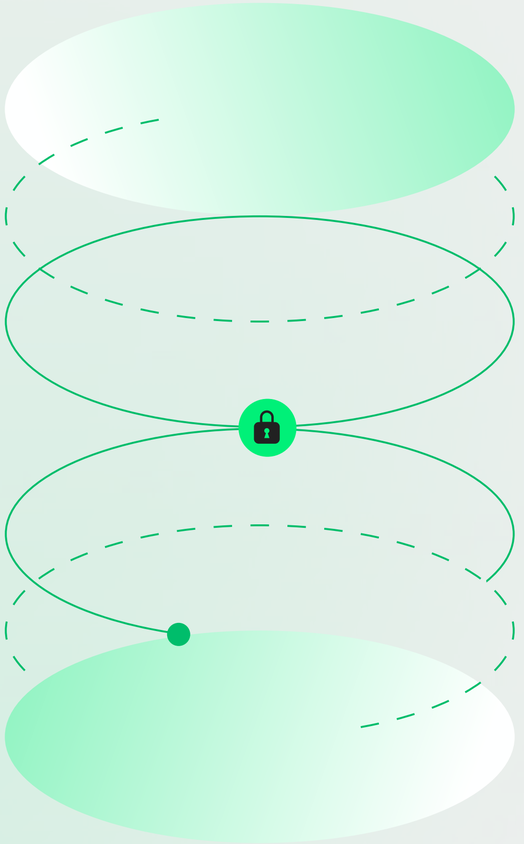
In other words, attackers discovered that the industry's widespread consumption of open source components comes with a considerable attack surface. A recently published taxonomy of attack vectors counts a total of **107 unique attack vectors**[49] [50], each representing a possible means to distribute malicious code to downstream consumers, application developers and end-users alike.

A study published in 2020 showed that typo-squatting (and comparable techniques) as well as account hijacking are among the most prominent attack vectors[51]. The former is still widely used in large attack campaigns comprising dozens and hundreds of malicious packages. Good news is typo-squatting attacks are quickly detected by now, also thanks to activity and popularity metrics, and account hijacking will diminish the more maintainers adopt 2FA.

While known vulnerabilities primarily matter for components that get deployed into production, it is important to note that **safeguards against supply chain attacks need to cover all dependencies** with access to critical project resources. Malicious test dependencies, for instance, may be able to change a package's content prior to its release - depending on the particular project setup and CI/CD pipeline.

Interestingly, controls useful for mitigating known-vulnerable dependencies, e.g., semver version ranges to facilitate the use of latest non-vulnerable releases, can be counterproductive in regards to supply chain attacks. Here, it is rather recommended to specify the exact version (so-called version pinning) to prevent the automated download of new, potentially compromised, releases.

Taking a look into the crystal ball, **we expect typo-squatting and dependency confusion attacks to continue** undiminished - it is just too easy to run large, automated attack campaigns. But we are more afraid of the **injection of intentional vulnerabilities into legitimitate packages**, which is more difficult to spot during merge request reviews than "active" malicious code used to, e.g., download and execute crypto miners. Last, we also believe that **new attack vectors will be discovered** - as in case of the trojan source[52] and dependency confusion attacks described in 2021.



# Program Analysis meets Dependency Lifecycle

This report is a result of long weeks of research. Rather than (only) looking for attention-grabbing headlines, we wanted to understand what's driving the biggest challenges for development and security teams today, and how those challenges can be mitigated. The most significant emerging challenges can be broken down into three main categories:

## Security noise

Today, the industry is focused on known vulnerabilities (CVEs) as an indicator of security. This has led software composition analysis (SCA) tools to drown developers in an endless stream of security alerts. After getting these alerts from security teams, developers must evaluate whether or not vulnerable code is actually reachable, or if the vulnerability is actually impactful. This slows down development considerably, as developers spend much of their time investigating and fixing vulnerabilities, and not writing value-adding code.

## Next-gen supply chain attacks

Most of the major supply chain attacks that have used OSS as their vector, or target, would not have been caught by looking at CVEs. Attacks like Typosquatting and Dependency Confusion target the maintainer, or the method in which OSS packages are consumed. In these cases, the focus on known vulnerabilities, while important, is not helping to enhance security.

## Maintenance is a nightmare

80% of code in modern applications is open source code, and as the report finds - 95% of vulnerabilities are found in transitive dependencies. Most security threats, including known vulnerabilities, lurk within the sea of transitive dependencies. The challenge is that developers rarely have visibility into their dependency tree, or how deep it goes.

Our conclusions on how to mitigate these issues are the same as the one that drove the core technology behind Endor Labs - Program analysis is required throughout the dependency management lifecycle. Program analysis can grant security and development teams a deep understanding of how code is actually being used. Without that understanding, teams will continue to struggle with the selection, security, prioritization, and maintenance of dependencies.

# Program Analysis meets Dependency Lifecycle

Below, we've outlined some of crucial building blocks that must be put in place to make life easier for security and development teams, and scale the use of open source at the enterprise:

**Detection and prevention of supply chain attacks**, e.g., the accidental use of typo-squatting packages or insecure CI/CD configurations, including the possibility to report on standards like NIST C-SRM or SLSA

**Complete software inventory** across the enterprise, supporting emerging standards like CycloneDX, and compliant with upcoming regulations


**Risk-based dependency selection** to reduce future operational and security risks

**Identification of bloated dependencies**, which can be safely removed in order to reduce build times, management overhead and the overall attack surface

**Monitoring of quality and security metrics**, specifically end-of-life packages (or package versions), which are not supported by the community any longer, thus, may receive security fixes at later points in time than supported ones... or never

**Update recommendations** that keep current API use into account in order to reduce the likelihood of regressions and developer effort

**Detection of known vulnerabilities** in direct or transitive project dependencies, followed by context-sensitive prioritization, which keeps into account whether vulnerable code can be reached in the context of a given application

# References


1 - https://www.enisa.europa.eu/publications/eucs-cloud-service-scheme

2 - https://en.wikipedia.org/wiki/Dependency_hell

3 - https://www.linuxfoundation.org/research/census-ii-of-free-and-open-source-software-application-libraries

4 - https://github.com/endorlabs/StateOfDependencyManagement2022

5 - https://github.com/ossf/criticality_score

6 - https://commondatastorage.googleapis.com/ossf-criticality-score/index.html

7 - https://openssf.org/community/alpha-omega/

8 - https://dl.acm.org/doi/10.1145/3368089.3409711

9 - https://chaoss.community/

10 - https://dl.acm.org/doi/10.1145/3239235.3240501

11 - https://bestpractices.coreinfrastructure.org

12 - https://owasp-scvs.gitbook.io/scvs/v5-component-analysis

13 - https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/

14 - https://riskexplorer.endorlabs.com/#/attack-tree?av=AV-602

15 - https://link.springer.com/chapter/10.1007/978-3-030-52683-2_2

16 - https://riskexplorer.endorlabs.com/#/attack-tree?av=AV-200

17 - https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

18 - https://azure.microsoft.com/en-us/resources/3-ways-to-mitigate-risk-using-private-package-feeds/

19 - https://checkmarx.com/blog/a-beautiful-factory-for-malicious-packages/

20 - https://www.usenix.org/system/files/sec19-zimmermann.pdf

21 - https://ieeexplore.ieee.org/abstract/document/7962360

22 - https://link.springer.com/article/10.1007/s10664-017-9589-y

23 - https://www.usenix.org/system/files/sec19-zimmermann.pdf

24 - https://ieeexplore.ieee.org/abstract/document/7962360

25 - https://www.linuxfoundation.org/research/census-ii-of-free-and-open-source-software-application-libraries

26 - https://ieeexplore.ieee.org/document/9506931

27 - https://semver.org/#spec-item-4

28 - https://ieeexplore.ieee.org/document/8721084

29 - https://ieeexplore.ieee.org/abstract/document/6975655

30 - https://dl.acm.org/doi/10.1145/3472811

31 - https://dl.acm.org/doi/abs/10.1145/3196398.3196401

32 - https://github.com/Netflix/archaius/tree/2.x

33 - https://commons.apache.org/proper/commons-text/security.html

34 - https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/

35 - https://dl.acm.org/doi/10.1145/3239235.3268920

36 - https://dl.acm.org/doi/abs/10.1145/2630069

37 - https://www.usenix.org/conference/usenixsecurity22/presentation/suciu

38 - https://unit42.paloaltonetworks.com/state-of-exploit-development/

39 - https://www.usenix.org/conference/cset20/presentation/householder

40 - https://republicans-oversight.house.gov/wp-content/uploads/2018/12/Equifax-Report.pdf

41 - https://dl.acm.org/doi/pdf/10.1145/3133956.3134072

42 - https://ieeexplore.ieee.org/document/6975655/

43 - https://arxiv.org/pdf/2110.07889.pdf

44 - https://link.springer.com/chapter/10.1007/978-3-030-68887-5_2

45 - https://dl.acm.org/doi/10.1145/3472811

46 - https://nvd.nist.gov/vuln/detail/CVE-2016-1000031

47 - https://bitbucket.org/snakeyaml/snakeyaml/issues/531/stackoverflow-oss-fuzz-47081

48 - https://incolumitas.com/data/thesis.pdf

49 - https://www.computer.org/csdl/proceedings-article/sp/2023/933600a167/1He7XSTyRKE

50 - https://riskexplorer.endorlabs.com/

51 - https://link.springer.com/chapter/10.1007/978-3-030-52683-2_2

52 - https://arxiv.org/pdf/2111.00169

53 - https://link.springer.com/article/10.1007/s10664-020-09914-8

54 - https://dl.acm.org/doi/abs/10.1145/3468264.3468589